This is a PythonQt program for a simple 3D robot simulator.

Requirements: python3, numpy, PyQt6, PyopenGL

1. The robot world consists of a 10m x 10m floor surrounded by 1m high walls.
2. Use a “left-handed” coordinate system, i.e. the origin is at the bottom left corner. Positive x is right, positive y is forward and positive z in up. NOTE - ROS uses a "right=handed" coordinate system
3. A left mouse click and drag rotates the camera view. The scroll wheel moves the view in and out.
4. The starting pose is camera is (5, -5, 10) and pointing down 45 degrees relative to the origin.
5. The world may have objects that are fixed and will cause collisions, and moveable objects that can be moved by the robot.
    - The function “create_object(Name, Type, Colour, FixedOrMoveable, (w,h,d), (x,y,z), (r, p, y))” has arguments:
        - Name is a unique name for the object
        - Type is the shape of the object, which can be: box, cylinder or sphere
        - Colour is the colour of the object, which can be Red, Yellow, Green or Blue
        - FixedOrMoveable can have values “fixed” or moveable.
        - (w, h, d) specify the width, height and depth of a box. For a cylinder, w is the diameter and h is the height, d is not required. For a sphere, only w is needed to specify the diameter of the sphere.
        - (x, y, z) specify the location of the centroid of the object in the world coordinates.
        - (r, p, y) specify the roll, pitch and yaw, where roll is a rotation about the y-axis, pitch is a rotation about the x-axis and yaw is a rotation about the z-axis. Using the left-hand convention, positive rotations are clockwise.
        - All objects must be supported, i.e. if the bottom of the object (in the z direction) is not in contact with the top of another, then the first object moves down until it comes into contact with a lower object or the floor
        - The surrounding walls are always fixed.
6. The model for the robot is a solid cylinder 60cm high and 50cm in diameter.
7. Each robot is equipped with a camera, a sonar array, a LiDAR.
    - The camera should deliver a 160x120 colour image in HSV format
    - The sonar should simulate 12 sensors distributed around the circumference of the cylinder. The beam width should be adjustable with a slider in a sidebar.
    - The LiDAR should return one distance measurement for each degree 0 to 359, in clockwise order.
8. Each robot has “magnet” fixed to the front of the robot. It can be represented by a black block 10cm square and 5 cm deep positioned 5cm above the floor.
9. When the magnet comes into contact with a moveable object, that object becomes attached to the robot and moves with the robot until a “detach” command is given.
10. There can be more than one robot, each one able to run concurrently with the others. Each robot should have a unique colour.
11. The display should have the main canvas that shows the robot world and a sidebar that will contain buttons and other displays.
12. A button in the sidebar should pop up a dialogue to get the name and load another Python file that is the “world file” that consists of a set of calls to the create_object function to initialise the objects in the world. The simulator may also be called with a command line argument that is the name of the world file.
13. Collision detection should prevent robots from passing through objects or other robots.
14. The simulation starts with one active robot on the floor, with the surrounding walls.
15. A button in the sidebar adds a new robot to the simulation. The robot must be placed so that it is not in collision with any other robot or obstacle.
16. The 0, 1, 2, ... keys select the active robot.
17. Once selected the robot’s camera view and LiDAR scan are shown in the sidebar.
18. The sidebar should also show a SLAM map derived from the LiDAR scans
19. The selected robot can be controlled manually by the A S W D keys
    - Holding the W key drive the robot forward, S drives it backwards. The A key turns the robot left, D turns it right
    - Releasing a key should stop the corresponding motion
20. The magnet is controlled manually "attach" and "detach" buttons in the sidebar. Attaching an object means that the object will move with the robot. Detaching an attached object will cause it to no longer move with the robot. A robot cannot attach to an object that is already attached to another robot.

21. The "Open Editor" button pops an editor window for the currently active robot.
22. The editor allows the user to write Python scripts that control that robot.
23. The editor has buttons for:
    - saving and opening Python scripts.
    - running and stopping the robot's program
24. Any compiler or runtime errors or output statements in the robot's Python script are shown in a console attached to the editor window.
25. The robot programs can call functions for controlling motors and accessing the sensors.

    - "drive(LinearVelocity, AngularVelocity)" sets the linear and angular velocities of the robot. LinearVelocity can range from -1.0 to +1.0, where 1 is the maximum velocity and a negative value drives backwards. Angular Velocity, also ranges from -1.0 to +1.0, where a negative value turns the robot left and positive turns it right. "drive(0.0, 0.0)" stops the robot.
    - A boolean function "holding()" returns true if a moveable object is attached to the robot via the magnet. It should return false otherwise.
    - A boolean function "close_enough()" returns true if the robot is close enough to a moveable object to be able to attach the magnet. It returns false otherwise
    - "attach()" activates the magnet, attaching a moveable object, if it is in range.
    - "detach()" detaches any object that is currently attached
    - "distance(A)" returns the LiDAR distance measurement at angle, A, where A is in the range 0 to 359, clockwise. A negative value of A is anticlockwise angle, i.e. A mod 360. A is circular, so if A=359, A+1 is 0.
    - "look()" returns a list of all the objects in the robot camera's field of view. The function returns a list of objects represented by an array containing the properties of the object: shape, colour distance and heading of the centre of the object relative to the centre of the robot, and the height and width of the bounding box of the object from the robot's camera's perspective.
        - "transform_to_map(Distance Angle)" transforms a point in polar coordinates, relative the centre of the robot, to the map's Cartesian coordinates.
    - "battery_level()" returns the percentage of battery level remaining.